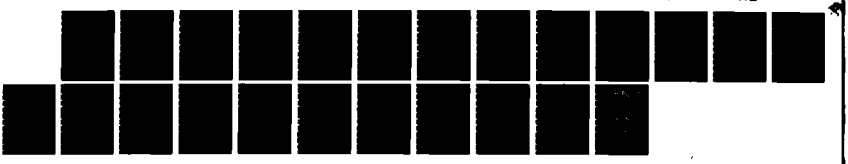
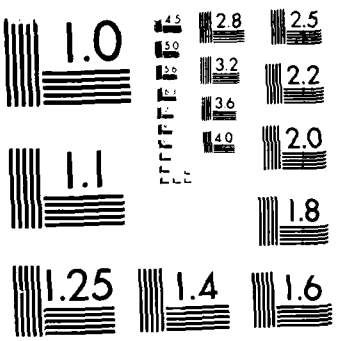
AD-A161 988 PARALLEL IMPLEMENTATIONS OF PRECONDITIONED CONJUGATE GRADIENT METHODS(U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE Y SAAD ET AL OCT 85 YALEU/DCS/RR-425 1/1

UNCLASSIFIED N00014-82-K-0184 F/G 9/2 NL





(10)

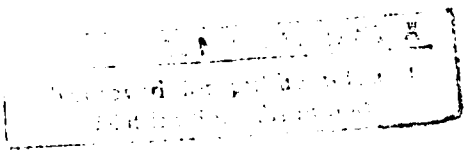
Parallel Implementations of Preconditioned Conjugate Gradient Methods

Youcef Saad and Martin H. Schultz
Research Report YALEU/DCS/RR-425
October 1985

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Parallel Implementations of Preconditioned Conjugate Gradient Methods

Youcef Saad and Martin H. Schultz
Research Report YALEU/DCS/RR-425
October 1985



Parallel Implementations of Preconditioned Conjugate Gradient Methods

Youcef Saad* and Martin H. Schultz*

Abstract. We consider a few different implementations of classical iterative methods on parallel processors with the purpose of studying how multiprocessor architecture affects performance. The framework is that of general nonsymmetric linear systems that arise from the discretization of partial differential equations and we concentrate on the solution methods based GMRES, a conjugate gradient-like method, combined with well-known preconditionings. The architectures considered are shared memory machines and loosely coupled linear or mesh connected arrays.

1. Introduction

One of the main issues in today's rapidly advancing parallel computing technology is to develop numerical algorithms for solving large-scale scientific problems that can take advantage of parallelism. There are basically two approaches towards achieving this goal. The first consists of attempting to discover new ways in which a given classical algorithm can be implemented in order to exploit its inherent parallelism. A second approach is to attempt to develop completely new algorithms, which may have no practical value in the one-processor case but are highly parallel and therefore attractive in a parallel environment. Optimal algorithm selection may depend on many machine parameters including the number of processors relative to the size of the problem to be solved. For example when solving a banded linear system, a parallel implementation of Gaussian elimination may be perfectly suitable if the number of processors is small relative to the bandwidth, but a different method, such as cyclic reduction, may be better otherwise, see [5, 7].

In this paper we will see these two approaches in action for the problem of solving general large sparse linear systems. We place ourselves in the framework of large nonsymmetric linear systems that arise from the discretization of partial differential equations. Among the best methods for handling these systems are the conjugate gradient like methods combined with some preconditioning techniques [2]. We have selected the GMRES algorithm [6] as the conjugate gradient type method and a simple ILU technique as the preconditioner. In fact our analysis can be extended to all conjugate gradient accelerators and to many of the well-known preconditionings techniques, as well as to the more traditional iterative methods like SOR and SSOR.

* Yale University, Computer Science Department, New Haven Connecticut. This work was supported in part by by ONR grant N00014-82-K-0184 and in part by a joint study with IBM/Kingston

One of the main difficulties in using preconditioning conjugate gradient type methods on parallel machines lies in the backward and forward solution algorithms associated with the preconditioner. When a natural ordering of the unknowns is used in the discretization, these triangular solves have the reputation of being very sequential in nature, although in reality it is only their classical implementation which is sequential. In the context of vector computers, this particular problem is a serious one and has been the subject of much recent research.

We will see that in a parallel computing environment, this difficulty can be handled in two different ways. First, one can use a pipelined version of the forward and backward solutions. For the natural ordering this can achieve a speed-up of the order of k, where k, the number of processors, does not exceed n, the number of grid points in one direction. To achieve higher speed-ups one must resort to a second approach based on different orderings of the unknowns. For illustration, we will examine two such orderings: a line red-black ordering and a natural red-black ordering. The degree of parallelism increases from the natural to the line red-black and natural red-black orderings thus allowing for higher speed-ups for the backward and forward solution algorithms. However, the overall efficiency of the preconditioned conjugate gradient method depends also on the quality of the preconditioning which is related to the ordering in a nonobvious way. By way of contrast, the SOR iterative method has the same rate of convergence for all three orderings. Nevertheless, experience suggests that for the preconditioned conjugate gradient type methods the gain in efficiency due to higher degree of parallelism will generally outweigh the loss incurred by a smaller convergence rate [1].

2. Preconditioned GMRES in parallel processing.

The GMRES algorithm [6] is an effective conjugate gradient like algorithm for solving general large sparse linear systems of equations of the form

$$Ax = f. \tag{2.1}$$

Assuming a preconditioner M is used on the left, see [2], we will be solving instead of (2.1), the preconditioned linear system

$$M^{-1}Ax = M^{-1}f. \tag{2.2}$$

A brief description of the preconditioned GMRES method follows. Details can be found in [6].

Algorithm : Preconditioned Generalized Minimal Residual Method (GMRES)

(1) *Start:* Choose x_0 and a dimension m of the Krylov subspaces.

(2) *Arnoldi process:*

- Compute $r_0 = M^{-1}(f - Ax_0)$, $\beta = ||r_0||$ and $v_1 = r_0/\beta$.
- For $j = 1, 2, .., m$ do:

$$h_{i,j} = (M^{-1}Av_j, v_i), \quad i = 1, 2, ..., j,$$

$$\hat{v}_{j+1} = M^{-1}Av_j - \sum_{i=1}^{j} h_{i,j}v_i$$

$$h_{j+1,j} = ||\hat{v}_{j+1}||, \quad \text{and}$$

$$v_{j+1} = \hat{v}_{j+1}/h_{j+1,j}.$$

Define H_m as the $(m+1)$ x m matrix whose nonzero entries are the coefficients h_{ij}.

(3) *Form the approximate solution:*

- Find the vector y_m which minimizes the function $J(y) \equiv ||\beta e_1 - H_m y||$, where $e_1 = [1, 0, \ldots 0]^T$, among all vectors of R^m.
- Compute $x_m = x_0 + V_m y_m$

(4) *Restart:* If satisfied stop, else set $x_0 \leftarrow x_m$ and goto 2.

Each outer loop of the above algorithm, i.e., the loop consisting of steps 2, 3, and 4, is divided in two main stages. The first stage is an Arnoldi step and consists of building a basis of the Krylov subspace K_m. The second consists of finding in the affine space $x_0 + K_m$ the approximate solution x_m which minimizes the residual norm. This is found by solving a least squares problem of size $m + 1$, whose coefficient matrix is upper Hessenberg.

For simplicity, we have omitted several details on the practical implementation in the above presentation. For example, in practice one computes progressively the least squares solution y_m in the successive steps $j = 1, \ldots, m$ of stage 2. Thus, at every step, after this least squares solution is updated, we obtain at no additional cost the residual norm of the corresponding approximate solution x_k without having to actually compute it, see [6]. This allows us to stop at the appropriate step.

Method	Multiplications	Storage
GCR(m-1)	[(3m+5)/2])N + NZ	(2m+1)N
ORTHODIR(m-1)	[(3m+5)/2])N + NZ	(2m+1)N
GMRES(m)	(m+3+1/m)N + NZ	(m+2)N

Table 1: Comparison of the costs of GMRES and GCR/ORTHOMIN.

Method	Steps	Mult./step
MILU(1)	21	24 N
ILU(1)	27	24 N
MILU(0)	51	20 N
ILU(0)	82	20 N

Table 2: Comparison of different preconditioners for SIAM Problem number 4.

The above algorithm is theoretically equivalent to GCR [2] and to ORTHODIR [4] but is less costly both in terms of storage and arithmetic [6]. Table 1 compares the cost of each outer loop of GMRES with the similar loops of its equivalent methods GCR and ORTHODIR. In the table NZ is the number of multiplications needed to perform the operation $M^{-1}Av$, for a given vector v. For large enough m, GMRES costs about 1/3 less than GCR/ORTHOMIN in arithmetic while storage is roughly divided by a factor of two. It can be shown that, in exact arithmetic, the method does not break down or, to be more accurate, that it breaks down only when it delivers the exact solution.

An important factor of the success of the conjugate gradient like methods is the preconditioning technique. In the classical incomplete factorization preconditionings, the matrix M is of the form $M = LU$ where L is a lower triangular matrix and U is an upper triangular matrix such that L and U have the same structure as the lower and upper triangular parts of A respectively, or may differ by one or a few diagonals. To solve a linear system with M requires a forward and a backward triangular system solution. Thus, each preconditioned GMRES does not cost much more than a typical non-preconditioned step. However, the number of iterations may be drastically reduced since $M^{-1}A$ is close in some sense to the identity matrix. Therefore, the rationale of the preconditioning techniques is that if one can reduce the total number of iterations substantially enough as compared to the incurred overhead, then the combination is economical. This is illustrated in Table 2 which shows the total number of iterations for two classes of preconditionings on the example SIAM Problem number 4, [3].

The above summarizes the situation for standard sequential machines. Consider now the implementation of the preconditioned GMRES algorithm on a parallel machine. There are mainly four types of computations in the algorithm.

1. matrix by vector multiplications;
2. forward and backward triangular system solutions when computing $M^{-1}Av_j$;
3. inner products;
4. linear combinations of vectors.

The operations 3 and 4 are highly parallelizable. Moreover, the most expensive part of the computation is in phase (2) of GMRES which builds the Arnoldi vectors by basically a modified Gram-Schmidt process. Much parallelism can naturally be exploited in this phase but we will skip the details. Operation 1 depends fundamentally on the structure of A, but is generally a parallelizable one.

The greatest apparent difficulty lies in the second type of computation: the classical implementations of the forward and backward solutions are highly ***sequential.*** We will show in the following sections how to handle this difficulty.

3. Architectures

There are essentially two modern approaches when building parallel computers. The first one, which we will refer to as the "big gun approach", is to build a tightly coupled machine consisting of a certain number of shared resources. The block diagram in Figure 1 illustrates such a machine which has the following features:

- k identical processors each having its own local memory;
- A global shared memory;
- A global broadcast bus.

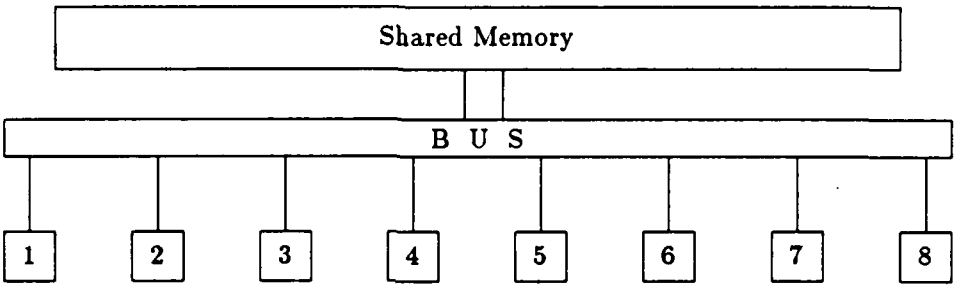


Figure 1: A Model of Global Shared Resources.

We will denote by B the bandwidth of the broadcast bus, and by b the bandwidth of each processor, both expressed in megawords per second. The speed of each arithmetic unit is denoted by s and is expressed in megaflops, i.e., in millions of floating point operations per second. It is usual that the ratio of the bandwidth B over b satisfies the condition

$$1 \leq \frac{B}{b} \equiv k^* \leq k,$$

which means that k^* processors can access the bus simultaneously. In this paper we will assume for simplicity that $B = b$, i.e., $k^* = 1$. This does not affect our qualitative results.

It is also desirable, but not always the case, that

$$s \geq b$$

which expresses that the arithmetic units are capable of processing data at the maximum rate allowed by their bandwidths.

A second approach to building modern parallel computers, is what we refer to as the " little gun approach", which consists of loosely connecting a number of small to medium size processors. The important difference with the former design is that there are no shared resources. Moreover, each processor is independent and makes control

decisions of its own. The synchronization is achieved by the availability of data: an operation is performed when the necessary operands have arrived from the neighboring processors. Examples include the linear array or ring represented in Figure 2 and the two-dimensional grid of processors represented in Figure 3. We assume that the grid is square with $\sqrt{k}$ processors on each side. In both cases we assume that each processor can send data *simultaneously* to any number of its nearest neighbors, at a rate of c megawords per second in *each* channel.

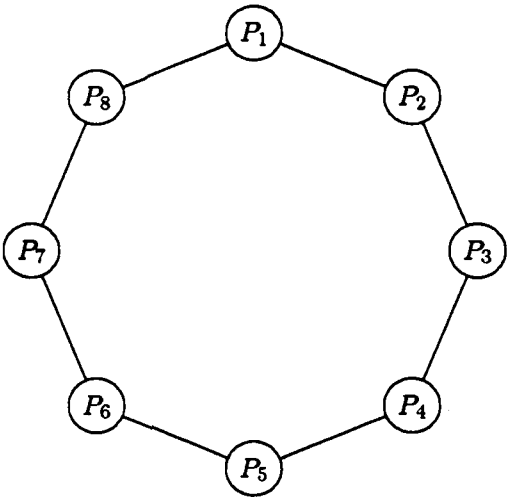


Figure 2: An 8-processor ring.

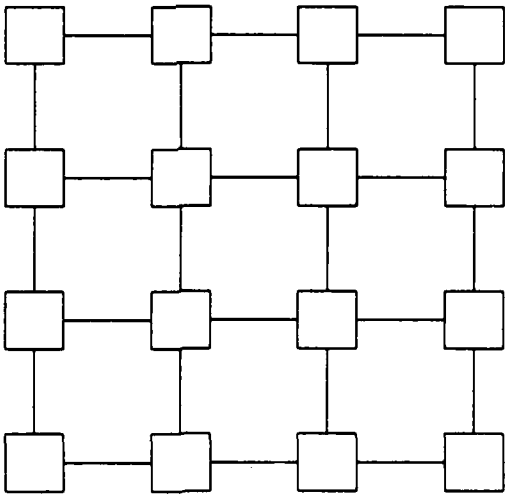


Figure 3: A 4 x 4 multiprocessor grid.

4. Implementing Conjugate Gradient Preconditionings on Multiprocessors

As we mentioned before, the conventional wisdom is that the main difficulty in implementing preconditioned conjugate gradient like methods on multiprocessors is the sequential nature of the forward and backward solutions involved in the preconditioning. In fact, we will show shortly that this is only a superficial problem, i.e., there is some inherent paralellism in the triangular system solutions that can be exploited. Let us consider the simplest preconditioner, namely the Incomplete LU preconditioner which we denote by ILU hereafter. For more details on how such an incomplete factorization is built the reader is referred to [2]. Here we will concentrate on the parallel implementation of the conjugate gradient like method using the ILU preconditioner, for example the combination GMRES/ILU.

Assume that the linear system $Ax = f$ arises from the discretization of a partial differential equation of the form

$$\mathcal{L}u = g$$

on the unit square [0, 1] x [0, 1] with, for example, Dirichlet type boundary conditions, where $\mathcal{L}$ is a non-self-adjoint elliptic partial differential operator.

It is customary to discretize the above equation using n interior points on each side of the square leading to a linear system of size $N = n^2$. Depending on the ordering chosen for the unknowns, we obtain different preconditionings. We will compare three such orderings in turn : the natural ordering, the line red-black ordering and the natural red-black ordering.

4.1. The natural ordering

For the natural ordering, the resulting 5-point discretization of the operator $\mathcal{L}$ is a block tridiagonal matrix A, with each diagonal block being an n x n tridiagonal matrix, and the codiagonal blocks being diagonal matrices. The corresponding incomplete factors L and U are lower and upper triangular matrices respectively with L having the same structure as the lower part of the matrix A. In the following discussion we will be concerned only with solving the lower triangular systems. The apparent difficulty in solving these triangular systems stems from the fact that in the usual organization of lower triangular system solvers the solution is obtained one coordinate at a time from top to bottom.

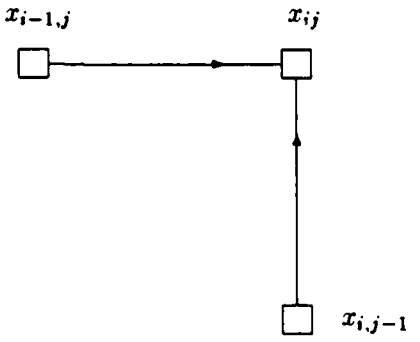


Figure 4: Stencil of the lower triangular matrix.

Some insight as to whether there exists better implementations is provided by considering the corresponding stencil on the grid, shown in Figure 4. The stencil establishes the data dependency between the unknowns in the lower triangular system solution. In this case, the dependency is a particularly simple one: as the arrows indicate, in order to compute the unknown in position (i,j) we only need the two unknowns in positions $(i-1,j)$ and $(i,j-1)$.

We consider at first a straightforward assignment of the data to the processors: the grid points are divided up into k equal blocks containing n/k lines each, and are assigned successively to processors $1,2,\ldots,k$, as is illustrated in Figure 6. From the stencil of L, it is clear that the grid points that are at the bottom boundary of the grid depend only on the unknown at their west, because of the Dirichlet boundary condition. Similarly, the points at the left boundary depend only on their southern neighbors. This suggests starting by computing x_{11} which does not depend on any other variable, and use this to get $x_{1,2}$ and $x_{2,1}$. Then these two values will in turn enable us to compute the elements $x_{3,1}, x_{2,2}$ and $x_{1,3}$. We can see the formation of a wave of computations. The subsequent steps of the wavefront algorithm can easily be deduced from the illustration of the four first steps in Figure 6.

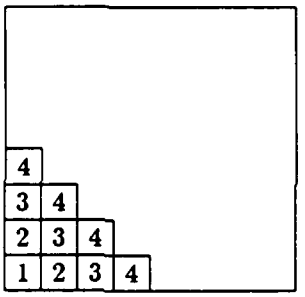


Figure 5: The first four steps of the block wavefront forward sweep

We note that with this algorithm the amount of parallelism is limited to at most n since simultaneous work can be performed only on the points of a same wave which has a maximum of n points. The consequence of this is that the time required to perform the forward solution with any version of this algorithm is at least $O(n)$, since the sequential time is of the order of $O(n^2)$.

To avoid too many communication start-ups, we group the steps and treat square blocks of size n/k at a time as is illustrated in Figure 5. Thus, in the first step we will determine all unknowns in the square block $x_{i,j}$, with $1 \leq i,j \leq n/k$. In step two we determine the unknowns $x_{i,j}$, in the square blocks in positions $(1,2)$ and $(2,1)$ of the block-grid and so on. A total of $2k-1$ such steps are required to complete this block wavefront algorithm, which corresponds to sweeping all antidiagonals. Each step consists of

- Data Transfer: sending a data packet of size n/k from some of the processors P_i to their neighbors P_{i+1};
- Arithmetic: computing one block of $(n/k)^2$ unknowns.

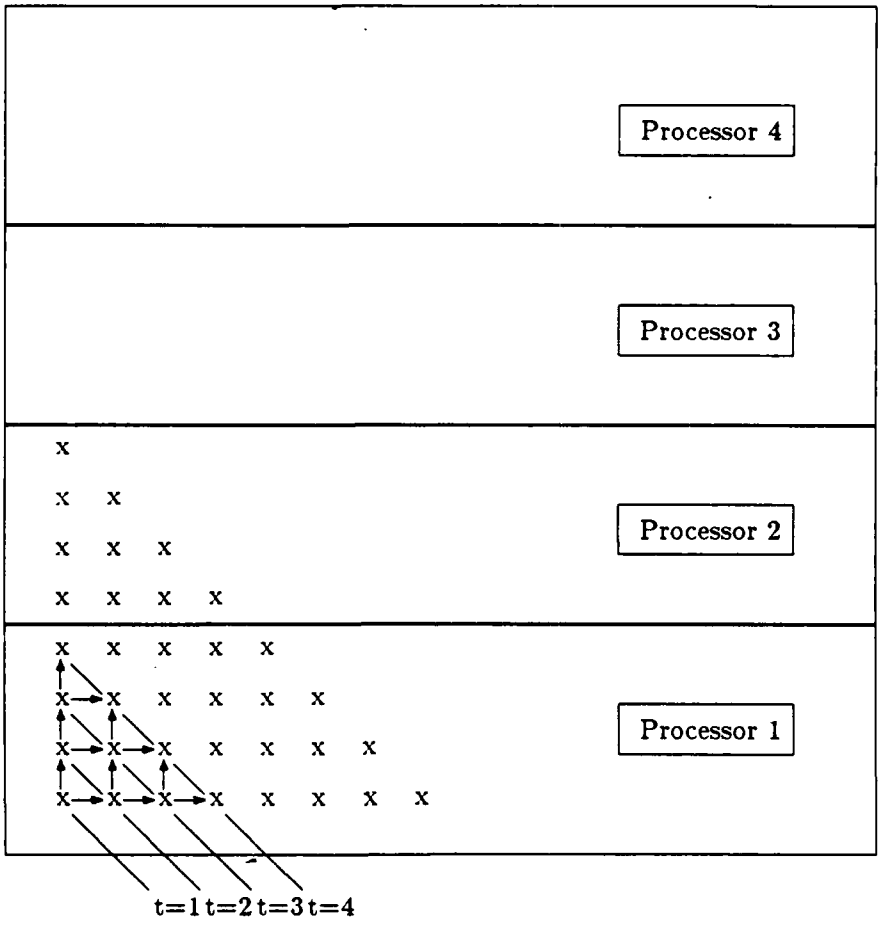


Figure 6: The wavefront approach for the forward sweep.

Let us consider the cost of arithmetic first. Since we assume that we do not overlap the successive steps of the above algorithm and that we do not overlap communication and arithmetic, at any given step, some of the k processors will be computing in parallel a square of n/k unknowns each. Hence the time to perform arithmetic is about

$$3\left(\frac{n}{k}\right)^2 \frac{1}{s}.$$

Clearly, the above arithmetic time does not depend on the architecture.

The communication time, on the other hand, clearly depends on the architecture. In the first step, we have processor 1 send a packet of length n/k to processor 2. Then in the second step we need to send a packet from processor 1 to processor 2 and another packet from processor 2 to processor 3. In general, in step $i, i \leq k-1$, we need to send a packet from processor j to processor $j+1$, for $j = 1, \ldots, i$. Let us first take the case of the shared resources model. Using the bus, the i^{th} step requires performing i data

transfers, each of which consists of moving a data packet of size n/k, i.e., each costs $\tau + \frac{1}{B}(n/k)$. Thus, for the shared resources model, the communication cost of the first $k-1$ steps is approximately

$$\sum_{i=1}^{k-1} i \left(\tau + \frac{n/k}{B} \right) \approx \frac{1}{2}\frac{n}{B}k + \frac{1}{2}k^2\tau$$

The timings for the steps $k, k+1, \ldots 2k-1$ are identical, except that they are summed in reverse order. Hence, the total time for the block wavefront algorithm using the bus is

$$t_S \approx \frac{n}{B}k + k^2\tau + 6\frac{n^2}{ks}.$$

It is interesting to determine the best achievable performance: given an arbitrary number of processors, we would like to know what is the best possible time in which the forward solution can be achieved. In other words we wish to minimize the above time with respect to the number of processors k. Differentiating the above function with respect to k and equating the result with zero we arrive at a third degree equation whose solution does not have a simple expression. However, it can be easily shown by using arguments similar to those in [7] that the optimal time which is obtained by neglecting the middle term in the above expression is a near optimum when n is large enough. This leads us to an optimal number of processors of the form $k_{opt} \approx \left(\frac{6B}{s}n\right)^{1/2}$ and an optimal time of the form

$$t_{opt,S} \approx 2\left(\frac{6}{Bs}\right)^{1/2} n^{3/2} + O(n)$$

The above result is disappointing in that the speed-up is limited to a factor of the order of $O(n^{1/2})$.

Looking at the ring or linear architecture, the only difference is that communication now takes place between neighboring processors and can be overlapped. Therefore, the communication time at each of the $2k-1$ steps simplifies into $\tau + \frac{(n/k)}{c}$ which brings the total time to

$$t_L \approx \frac{2n}{c} + 2k\tau + 6\frac{n^2}{ks}.$$

One might ask again what is the optimal number of processors and the optimal time. The answer here is clearly that

$$k_{opt} = \sqrt{\frac{3}{s\tau}}n$$

and

$$t_{opt,L} = \left[\frac{2}{c} + 4\sqrt{\frac{3\tau}{s}}\right] n.$$

In other words a linear time can be achieved with a linear array or a ring.

We have shown that by a simple reorganization of the forward solution algorithm, consisting of pipelining in a wavefront approach, a speed-up of the order of n can be achieved. The wavefront algorithm has an inherent parallelism of at most n which is precisely the largest width of the waves. Unfortunately, this means that there is no other implementation on any type of architecture that will achieve a speed-up higher than $O(n)$. For this reason we do not attempt to show an implementation of this algorithm for the grid architecture. In case the processor grid has a number of processors that does not exceed n, then a simple method is to map the 2-D grid into a linear array and use the wavefront algorithm. Otherwise, if $k>>n$, we must resort to alternative algorithms that can attain higher degrees of parallelism. This is our goal for the following sections.

4.2. The line red-black ordering

In the line red-black ordering we color the lines of unknowns alternatively in red and black starting from the bottom, see Figure 7, where the red points are represented by a cross and the black ones by a bullet. We then number the unknowns of the red lines first, from bottom to top, and then the black unknowns from bottom to top.

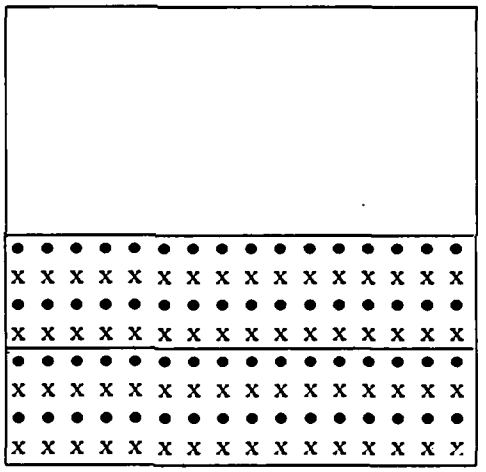


Figure 7: Linear array assignment for the line red-black ordering

As is revealed by the corresponding L matrix shown in Figure 8, the first $N/2$ unknowns, i.e., the red unknowns, can be solved for independently of the black points. Indeed they form a decoupled linear system of size $N/2$, consisting of $n/2$ (independent) bidiagonal systems. In Figure 7, this means that all the bidiagonal systems associated with the red points can be solved for first, then their data is used for solving for the black points.

Assume that each processor contains $\frac{n^2}{2k}$ red points and $\frac{n^2}{2k}$ black points. Considering arithmetic alone, we can solve for the red points in time $\frac{n^2}{ks}$ which corresponds to solving $n/2$ bidiagonal systems in parallel, each of which requires $2(n/k)$ arithmetic operations. To get the black points we need to subtract from the second half of the right hand side, the product of the block $(2,1)$ of the matrix L as shown in Figure 8 by the vector of

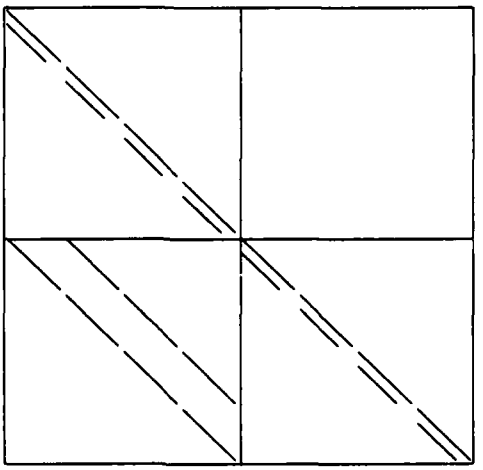


Figure 8: The line red-black L matrix

the red unknowns. Then we solve the bidiagonal systems in the same way as for the red points. This results in a total of $\frac{2n^2}{ks}$ for this second phase and a total of

$$3\frac{n^2}{ks}.$$

For communication, we need to move n red points across each boundary in one of the directions north or south. For example, in Figure 7, each processor number j must move n red data to its neighbor $j-1$ which holds the grid points of the partition located at the south of that held by processor j. For the shared resources model this requires the time

$$k\left(\tau + \frac{n}{B}\right)$$

resulting in an overall estimated time of

$$t_S = 3\frac{n^2}{ks} + k\tau + \frac{kn}{B}.$$

The best achievable time is again worse than linear in n:

$$k_{opt} = \frac{n}{\sqrt{\frac{s}{3}(\tau + \frac{n}{B})}}$$

and

$$t_{S,opt} = 2n\sqrt{\frac{3}{s}\left(\tau + \frac{n}{B}\right)}$$

Let us now take up the case of the ring. Here the communication of the n red unknowns in a given direction of the ring can be overlapped: all processors can send data in one of the two directions of the ring in parallel. This requires the time

$$\tau + n/c.$$

Hence the total time for the ring is,

$$t_R = 3\frac{n^2}{ks} + \tau + \frac{n}{c}.$$

The best achievable time is for $k_{opt} = n$ and results in

$$t_{R,opt} = 3\frac{n}{s} + \tau + \frac{n}{c}.$$

Observe the big difference in the coefficient τ of the communication start-up with that of the natural ordering.

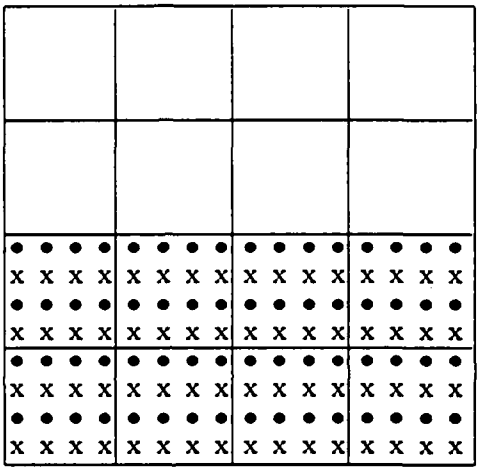


Figure 9: Mesh to processor grid assignment for the red-black ordering.

Next we examine the case of the two-dimensional grid. The grid points are assigned in a manner analogous to the previous case: we map the region naturally onto the processor grid as is shown in Figure 9. An important observation is that each of the $n/2$ independent bidiagonal systems of size n is now distributed among $\sqrt{k}$ processors, located on the same row of the grid of processors. If a standard forward sweep algorithm were to be used, it would not be possible to achieve a time better than $O(n)$, because this algorithm has intrinsically n steps that are sequential. Note that a similar barrier problem was encountered in the context of ADI methods in [8]. The remedy considered in [8] is to resort to substructuring.

The substructuring algorithm can be briefly described as follows. Assume that we want to solve *one* lower bidiagonal system $Fx = b$ of size n in κ processors, each of which holds a partition of n/κ successive rows of the system. Here the $\kappa \equiv \sqrt{k}$ processors constitute a row of the grid of processors. According to our assumptions these κ processors form a ring. We distinguish three phases in the substructuring algorithm. The first phase is a variant of Gaussian elimination within each partition, which involves no interprocessor communication. We perform a forward sweep in each processor j, $j = \{1, 2, ..., \kappa\}$, on equations $(j-1)\frac{n}{\kappa} + i$, $i = \{2, 3, ..., \frac{n}{\kappa}\}$, in order to

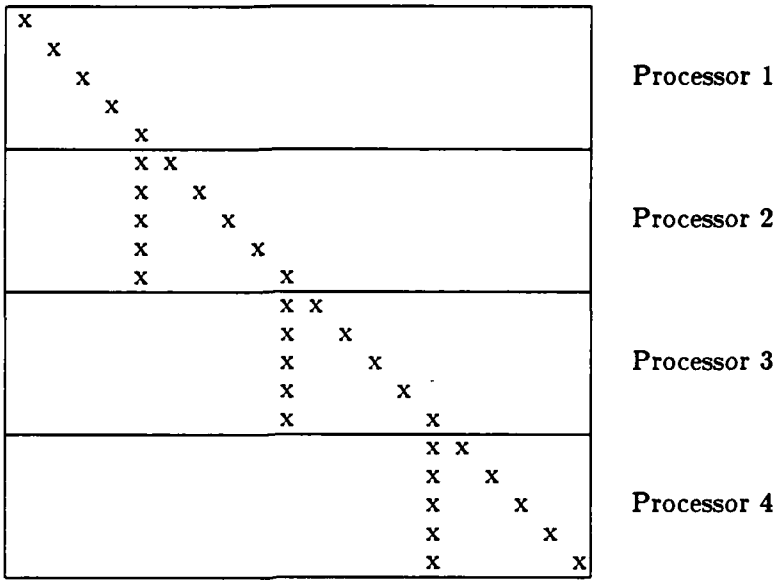


Figure 10: A substructuring lower triangular system solution on four processors.

eliminate the sub-diagonal elements within each $\frac{n}{\kappa}$ x $\frac{n}{\kappa}$ diagonal block of the bidiagonal matrix F. The result of the first phase is illustrated in Figure 10 for a system of size 20 distributed in $\kappa = 4$ processors. There are 6 arithmetic operations for each step of the elimination, leading to a total of

$$t_1 = \frac{6n}{\kappa s}. \tag{4.1}$$

There is no communication involved in this first phase.

Observe that the unknowns $j\frac{n}{\kappa}$, $j = \{1, 2, ..., \kappa\}$, satisfy an independent bidiagonal system of κ equations distributed with one equation per processor. In the second phase we will solve for these unknowns. Since the processors form a ring the total cost, using a standard forward sweep algorithm is roughly

$$t_2 \approx (\kappa - 1)(\tau + \frac{1}{c} + \frac{3}{s}). \tag{4.2}$$

In the third and final phase of the substructuring algorithm, each processor j solves for the other variables by subtracting multiples of the fill-in columns. This operation involves no communication, except for the transfer of the variables $j\frac{n}{\kappa}$ already computed in phase 2 from processors j to processors $j + 1$ for $j = \{1, 2, ..., \kappa - 1\}$. Therefore, the cost of the third phase on a linear array or ring is approximately

$$t_3 \approx \frac{3n}{\kappa s} + \tau + \frac{1}{c}. \tag{4.3}$$

We now describe how to implement the substructuring algorithm to the forward sweep of the preconditioning. We consider the application of this technique for solving

for the red points first. In the first pass of the preconditioning all the rows of the processor grid work simultaneously, with no communication between the processors of different rows. There are $\frac{n}{2\kappa}$ independent bidiagonal systems in each row of processors, each distributed among the κ processors of the row. Implementing the first phase is straighforward and its cost is

$$t_1' = \frac{n}{2\kappa}\frac{6n}{\kappa s} = \frac{3n^2}{ks} \tag{4.4}$$

Implementing the second phase requires more care. We have $\frac{n}{2\kappa}$ independent bidiagonal systems, each of size κ, in each row of processors distributed with one equation per processor. A straight-forward implementation consists in sweeping for all variables of the first column of subsquares before forward-sweeping in the second column of subsquares and so on. However, this is inefficient because only one processor of each row will be active at any time. In fact, this naive approach would require solving successively $\frac{n}{2\kappa}$ bidiagonal systems of size κ each and therefore the time would again be nearly linear in n. The remedy is pipelining. Processors $(*,1)$ start sweeping with only one bidiagonal system each. When the right boundaries of the subsquares are reached, each processor $(*,1)$ starts solving a second linear system. Meanwhile processor $(*,2)$ of the second column takes over the forward sweep of the first system and so forth. After every n/κ elementary sweeps, we start processing a new system in each processor $(*,1)$. The last system is started after $\frac{n}{2\kappa}$ elementary steps. To complete the sweep of each last system in each row of processors, we need an additional $\kappa - 1$ steps. Therefore, we need a total of $\frac{n}{2\kappa} + \kappa - 1$ successive elementary steps, each costing t_2, as given by (4.2). The total time for this second phase is approximately

$$t_2' = (\frac{n}{2\kappa} + \kappa - 1)\left(\tau + \frac{1}{c} + \frac{3}{s}\right). \tag{4.5}$$

The third phase is fully parallelizable and requires a time of

$$t_3' \approx \frac{3n^2}{2ks} + \tau + \frac{n}{2\kappa c}. \tag{4.6}$$

Note that we need only one communication start-up by sending all the required unknowns at once in the third phase, which is why the term τ in (4.3) is not multiplied by $\frac{n}{2\kappa}$, unlike the other terms.

The above includes only the cost of solving for the red unknowns. To solve for the black unknowns, each processor needs first to receive $n\kappa$ red points from its northern neighbor. As can be easily seen, with our assumption that communication can be overlapped in all channels of a node, this data movement operation can be accomplished in one pass in which $n\kappa$ data are moved in parallel. Thus the communication time is of the order of

$$t_4' = \tau + \frac{n}{\kappa c}.$$

Each black unknown is coupled with two red unknowns which have been already calculated. We then need to remove these two unknowns from every black equation by

subtracting multiples of them from the the corresponding parts of the right hand sides. This requires a total time of

$$t_5' = \frac{n}{2\kappa}\frac{n}{\kappa}\frac{4}{s} = \frac{2n^2}{ks}.$$

Finally, we now obtain a system for the black points which is similar to the one we had for the red points before: $\frac{n}{2\kappa}$ independent bidiagonal systems in each row of processors. We can solve these equations by substructuring at the cost of (4.6). The grand total comes to $t_G = 2(t_1' + t_2' + t_3') + t_4' + t_5'$, or,

$$t_G = \frac{11n^2}{ks} + \left(\frac{n}{\kappa} + 2\kappa + 1\right)\tau + \left(\frac{3n}{\kappa} + 2\kappa - 2\right)\frac{1}{c} \tag{4.7}$$

In order to minimize the above time with respect to k we rewrite it as

$$t_G(k) \approx \frac{11n^2}{ks} + (\tau + \frac{3}{c})\frac{n}{\kappa} + (2\tau + \frac{2}{c})\kappa + Constant$$

which is of the form

$$t_G(k) = \alpha\frac{n^2}{k} + \beta\frac{n}{\sqrt{k}} + \gamma\sqrt{k} + Constant.$$

The *approximate* minimum of the above function of k is achieved for

$$k_* = \left(2\frac{\alpha}{\gamma}n^2\right)^{2/3}$$

and has the value

$$t_{G,*} \approx \frac{3}{2}\left(2\gamma^2\alpha\right)^{1/3} n^{2/3} = 3\left[\frac{11}{s}\left(1 + \frac{1}{c}\right)^2\right]^{1/3} n^{2/3}. \tag{4.8}$$

An argument similar to one used in [7] shows that that $t_{G,*}$ is close to the actual minimum of $t_G(k)$ when n is large.

We point out that a similar substructuring method was used for the alternating direction method and has led to a very similar result, namely that each step of the ADI iteration could be achieved in time $O(n^{2/3})$ on a square grid of processors.

4.3. The natural red-black ordering

A simple way of achieving a higher degree of parallelism is to use an incomplete factorization of the matrix obtained from the red-black ordering of the unknowns: the unknowns are colored alternatively red and black so that there are no two neighboring points of the same color and then the points are naturally numbered starting with the

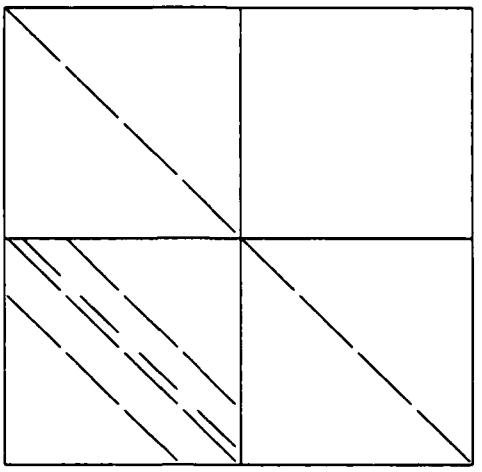


Figure 11: The red-black L matrix

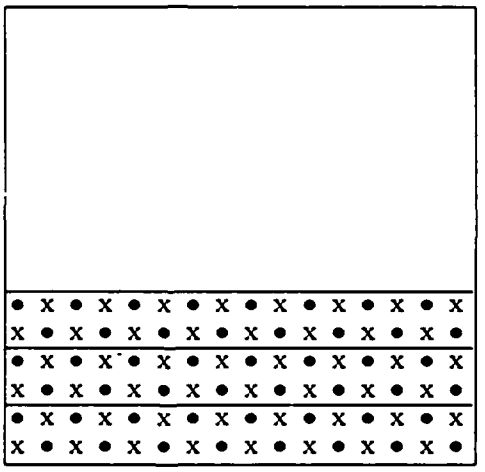


Figure 12: Mesh to ring assignment for the red-black ordering

black ones and then the red ones. In effect this amounts to permuting the rows and columns of the original matrix and then performing the incomplete factorization.

An important feature of the red-black incomplete LU factorization is that the red points do not depend on the black ones. Therefore, the red unknowns can be solved for in a trivial way. This can be easily understood by a look at the corresponding L matrix shown in Figure 11 for a model example: the first $N/2$ unknowns form a decoupled linear system which is diagonal. Therefore, one can hope for a speed-up of the order of $O(n^2)$.

The forward sweep can be organized in a simple way as follows. First we must solve the diagonal system corresponding to the first $n^2/2$ unknowns, i.e., the red unknowns. Then we move some of these unknowns where they are needed and solve for the black

unknowns. This last phase requires forming a linear combination of the 4 red unknowns that are neighbors to a given black point and dividing by some coefficient.

With the assignment of Figure 12, it is clear that we must send a distinct data packet of size $n/2$ from each processor P_i to each of its neighbors P_{i+1} and P_{i-1}. The time for executing the arithmetic is approximately $\frac{n^2}{2ks}$ for the red points solution (a diagonal solve) and $\frac{5n^2}{2ks}$ for the black points solution, which comes to a total of $\frac{3n^2}{ks}$.

Concerning communication for the shared memory model, we need k successive data transfers and therefore, the communication time for the shared memory model is approximately $k(\frac{n}{B} + \tau)$. Hence, the total time required for a forward solution in the case of the red-black ordered incomplete factorization is approximately

$$t_{RB,P} \approx k\left(\frac{n}{B} + \tau\right) + \frac{3n^2}{ks}.$$

The above time reaches its minimunm when k is of the order of $k_{opt} = O(n^{1/2})$ and is the minimum is approximately $t_{opt} = O(n^{3/2})$.

For a linear array or a ring, the only difference is that we can overlap communication between successive processors when moving boundary red points to the neighboring processors. As a result the total communication time simplifies into $2(\frac{n}{2b} + \tau)$, i.e., the total time is roughly:

$$\frac{n}{b} + 2\tau + \frac{3n^2}{ks}.$$

Observe that the above time will always be decreasing as k increases. However, since we cannot use more than n processors in this scheme, the optimal time is actually reached for the maximum allowable number of processors, i.e., for $k_{opt} = n$ and the corresponding time is simply $t_{opt} \approx \frac{n}{b} + 2\tau + \frac{3n}{s} = O(n)$.

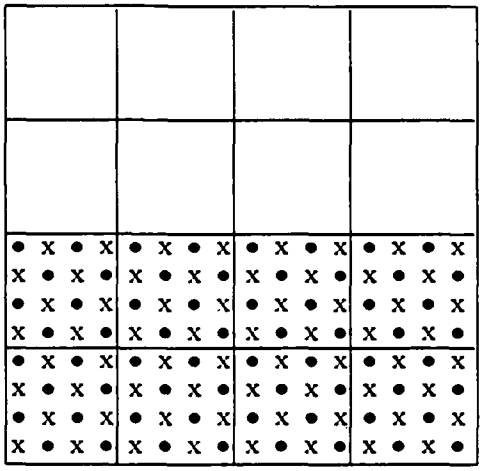


Figure 13: 2-D array assignment for the red-black ordering

We now consider a two-dimensional grid of $\sqrt{k}$ x $\sqrt{k}$ processors and assign the grid points in a simple manner by mapping the region naturally onto the processor grid as is shown in Figure 13. In this situation, the communication demand is reduced by the fact that we now have to transfer a data packet of size only

$$\frac{1}{2}\frac{n}{\sqrt{k}},$$

to each neighboring processor. Therefore, the above communication time becomes

$$2\left(\frac{n}{2c\sqrt{k}} + \tau\right),$$

and the total estimated time for performing the forward solve is approximately

$$t_{RB,2-D} \approx \frac{n}{c\sqrt{k}} + 2\tau + \frac{3n^2}{ks}.$$

An important new feature of the above formula is that it is decreasing as k increases and reaches its minimum value when k is the maximum possible number of processors for which this implementation is possible, i.e., $k_{opt} = n^2$. Then the optimal time becomes simply $t_{opt} \approx 1/c + \tau + 3/s$ which is independent of n. Of course in a sense this is deceiving since we have used $O(n^2)$ processors.

5. Summary and Conclusion

In preconditioned conjugate gradient type methods, one needs to solve a system of the form $My = z$ at every step, where M is usually in factored form $M = LU$. We have shown a few different ways of implementing these solves on three model architectures. The usual incomplete factorizations without extra fill that are associated with natural orderings of the unknowns offer some degree of parallellism that allows us to speed-up these solves by a factor of $n^{1/2}$ for shared global resources and n for rings, where n is the number of unknowns in each direction. To achieve the higher degrees of parallelism that we would hope to see with nearest-neighbor multi-dimensional architectures, we need a different ordering of the unknowns before performing the incomplete factorization. Thus, using a line red-black ordering, we can solve the above systems in time $O(n^{2/3})$ on a two-dimensional processor grid, while by using the natural red-black ordering we can solve them in constant time.

However, we should point out that this is only part of the story in that we have not considered the impact of different orderings on the rates of convergence of the corresponding iterative processes. The rates of convergence are difficult to study from the theoretical point of view. Not only is it difficult to get any information on the spectral condition numbers of the preconditioned matrix $M^{-1}A$, but even if these were available they would be of little help, since the rate of convergence depends on the global eigenvalue distribution more than anything else. Experience shown elsewhere [1] indicates that, at least for the red-black ordering, the loss in convergence is limited. Moreover, for the more traditional iterative methods such as SOR, the rate of convergence is independent of the ordering.

Another aspect to which we have devoted little attention is that in the overall cost of one step of the preconditioned conjugate gradient method we must add to our timings of the preconditioning part a term of the form $O(n^2/k)$ which accounts for the various other operations as was described in section 2. We have argued in Section 2 that these other parts are highly parallelizable and have ignored the question. In reality, this is not entirely correct when it comes to considering a large number of processors. The corresponding formulas which estimate the runing time have to be revised taking into account which of the various versions of conjugate gradient method we are using. Moreover, another difficulty is that for very large k the inner products cannot be fully parallelizable and may yield a deteriorated time of the form $O(\frac{n^2}{k} log_2\ k)$. Our goal was to show that what is traditionally viewed as a major bottleneck in these methods, namely the solution of the preconditioning systems $My = z$, was amenable to parallel implementation.

A comparison of the performances of the three models shows that the two- dimensional grid of processors allows us to exploit parallelism more efficiently and suffers less from the impact of communication delays. Clearly, the reason for this is that the problem we are considering is very much based on nearest neighbor interaction and maps perfectly onto a grid of processors. This is a common situation with all problems arising from the discretization of partial differential equations.

References

[1] D. Baxter, *Personal communication,* 1985.

[2] H.C. Elman, *Iterative Methods for Large Sparse Nonsymmetric Systems of Linear Equations,* Ph.D. Thesis, Yale University, Computer Science Dpt., 1982.

[3] H.C. Elman, *Presentation given at the mini-symposium on iterative methods, this meeting.,* 1985.

[4] A.L. Hageman and D.M. Young, *Applied Iterative Methods,* Academic Press, New York, 1981.

[5] L. S. Johnsson, *Fast Banded Systems solvers for Ensemble Architectures.,* Technical Report YALEU/DCS/RR-379, Computer Science Dept., Yale University, 1985.

[6] Y. Saad, M.H. Schultz, *GMRES: a Generalized Minimal residual algorithm for solving nonsymmetric linear systems,* Technical Report 254, Yale University, 1983. to appear in SISSC.

[7] Y. Saad, M.H. Schultz, *Direct parallel methods for solving banded linear systems,* Technical Report YALEU/DCS/RR-387, Computer Science Dept., Yale University, 1985.

[8] S.L. Johnsson, Y. Saad, M.H. Schultz, *The alternating direction algorithm on multiprocessors,* Technical Report YALEU/DCS/RR-382, Computer Science Dept., Yale University, 1985.

END

FILMED

2-86

DTIC